TECHNICKÁ UNIVERZITA V LIBERCI
FAKULTA MECHATRONIKY, INFORMATIKY
A MEZIOBOROVÝCH STUDIÍ

BAKALÁŘSKÁ PRÁCE

LIBEREC 2012 **MATĚJ NEBŘENSKÝ**

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 Informační technologie

Studijní obor: Informační technologie

PŘEKLADAČ VYŠŠÍHO PROGRAMOVACÍHO JAZYKA

HIGH-LEVEL PROGRAMMING LANGUAGE COMPILER

Matěj Nebřenský

Vedoucí bakalářské práce: Ing. Tomáš Martinec Ph.D.

Konzultant bakalářské práce: Ing. Přemysl Svoboda

Rozsah práce:

Počet stran textu....26

Počet obrázků.....1

Počet tabulek.....0

Počet grafů.....0

Počet stran příloh. .5

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Akademický rok: **2011/2012**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Matěj Nebřenský**
Osobní číslo:	**M08000153**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Překladač vyššího programovacího jazyka**
Zadávající katedra:	**Ústav mechatroniky a technické informatiky**

Zásady pro vypracování:

1. Seznamte se s konstrukcí moderních překladačů vyšších programovacích jazyků.
2. Navrhněte vlastní programovací jazyk, určený pro zvolenou platformu.
3. Pomocí vhodných nástrojů vytvořte takový překladač a otestujte ho na jednoduchých příkladech

PROHLÁŠENÍ

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložil na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

V Liberci dne 18. května 2012

.....
Podpis

PODĚKOVÁNÍ

Rád bych na tomto místě poděkoval panu Ing. Tomáši Martincovi, Ph.D. jak za zadání velmi zajímavého i když poněkud rozsáhlejšího tématu, tak za všechny odborné rady a názory. Dále děkuji všem vyučujícím za vše, co mě stihli naučit. V neposlední řadě musím též vyjádřit vděčnost své přítelkyni Sylvii Peťkové za bezmeznou trpělivost a samozřejmě rodině za nekonečnou podporu při studiu na vysoké škole.

ANOTACE

Následující text obsahuje kompletní dokumentaci k překladači TulToAsm51, včetně vysvětlení všech principů a pravidel jazyka TUL - Temporary Universal Language navrženého s ohledem na dobré zkušenosti s jednoduchým programováním v jazycích Pascal a Matlab.

TulToAsm51 zpracovává složky se vstupními soubory *.tul s obsahem odpovídajícím pravidlům gramatiky jazyka TUL uvedeným v kapitole TUL kód. Rozliší v něm všechny známé symboly, z těch nechá vyrůst derivační strom a ten poté převádí do kódování Assembleru 8051 včetně přiřazení fyzických adres všem proměnným. Vygenerovaný soubor *.asm lze použít s jakýmkoli softwarem na linkování a nahrávání *.asm souborů do libovolného PLC s procesorem Intel MCS-51.

Překladač je vystavěn na platformě Java jako přenositelný mezi operačními systémy Linux a Windows (teoreticky i OS-X a Android). Jedná se o konzolovou aplikaci bez požadavku na uživatelský vstup, která se snadno implementuje do univerzálních vývojových prostředí jako Geany (pro Linux) či PS-Pad (pro Windows).

Tato práce vychází z poznatků zaznamenaných v předcházejícím bakalářském projektu Teorie tvorby překladačů vyšších programovacích jazyků. Vývoj kvalitního překladače však obnáší kromě důkladné teoretické přípravy i dlouhá léta pečlivého ladění, tudíž dveře k úpravám a doplnění chybějících funkcí zůstávají otevřeny.

KLÍČOVÁ SLOVA:

Scanner, parser, compiler, lexikální analýza, syntaktická analýza, symbol (terminál), neterminál, derivační strom, výraz, pin, port, 8051 (x51, 51).

ANNOTATION

Following text contains the complete documentation of TulToAsm51 compiler, including definitions of all TUL - Temporary Universal Language - principles and rules. This language refers to good experiences with easy programming in Pascal and Matlab.

TulToAsm51 can run with *.tul input files containing code according to TUL grammar rules caught by chapter TUL coding. It recognizes all known symbols and lets a derivation tree grow up from them. This tree is then compiled to Assembler 8051 language including variables' physical addresses assignment. Generated *.asm file can be used with any linking and loading software for PLCs based on Intel MCS-51 processor.

This compiler is build on Java platform and so it is portable between operation systems Linux and Windows (probably even OS-X and Android). TulToAsm is designed as a console application without any run-time user input requirement, so it is easy to implement it in some of universal IDEs like Geany (for Linux) or PS-Pad (for Windows)

This thesis is based on the insights captured in foregoing bachelor project Theory of higher programming language parser. However, high quality compiler development requires a very long time for testing and debugging, so the door to any reconstructions or adding missing functions are still opened.

KEY WORDS:

Scanner, parser, compiler, lexical analysis, syntactic analysis, symbol (terminal), non - terminal, derivation tree, expression, pin, port, 8051 (x51, 51).

Obsah

SEZNAM POUŽITÝCH ZKRATEK

PLC - Programmable Logic Controller – malé programovatelné logické zařízení

PAC - Programmable Automation Controller – velké programovatelné logické zařízení s podporou řízení průmyslových sběrnic

RAM - Random Access Memory, není-li řečeno jinak, je míněna vnitřní RAM procesoru

LSB - Nejméně významný bit (Least Significant Bit)

MSB - Nejvýznamější bit (Most Significant Bit)

TUL - Temporary Universal Language

1 ÚVOD

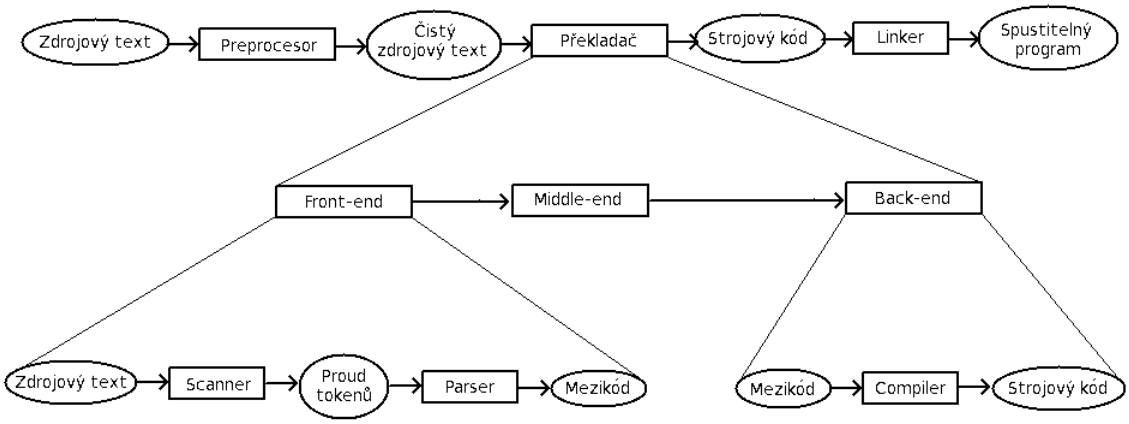
Aby mohl být uživatelem pracně vytvořený zdrojový kód kompletně vykonán a tudíž nepostrádal v reálném světě smysl, potřebuje kromě fyzického stroje ještě software, který je schopen zdrojový kód vysvětlit počítači tak, aby ho pochopil co možná nejsprávněji. Mimo to samozřejmě existují i jiné zajímavé platformy než PC, třeba roztomile malinké PLC od Atmelu osazené čipem Intel MCS-51. Téma překladačů pro jiné procesory než pro všeobecně známý x86 se stává v současné době velmi populární. Ve světě nepřeberného množství dostupných vzájemně nekompatibilních programovatelných zařízení, kterým málokdo rozumí, se zkušenosti s vývojem mechanismů převádějících oblíbené instrukce, jako přiřazení pomocí =, cyklus `for` atd. na změť hexakódů, kterou přesně potřebuje ten který procesor, vyvažuje zlatem.

Naprogramovat celý takový chytrý překladač však znamená běh na dlouhou trať. Proto celý bakalářský projekt zaplnila jen nutná teoretická příprava a první příkazy v java-kódu vznilky až v rámci bakalářské práce. V žádném případě si tato neklade nároky na implementaci všech standardních funkcí a perfektní odladění. Cílem snažení je napsat jakýkoliv smysluplný program v definovaném jazyce TUL, ten přeložit překladačem TulToAsm51 do Assembleru 8051, cizím linkerem převést do hexa-kódu a nahrát do školního přípravku. Z tohoto důvodu dostává i navržený jazyk honosné jméno TUL - Temporary Universal Language. Znamená to, že bude obsahovat jen pár příkazů, se kterými už lze napsat kupříkladu kalkulačku na jeden výstupní, dva vstupní porty a dva vstupní piny pro čtyři základní aritmetické operace.

Jelikož přístroje koncových uživatelů překladače TulToAsm51 mohou hostit různé operační systémy, jedná se o multiplatformní aplikaci v jazyce Java. Nicméně podle nejlepších zkušeností bylo zvoleno vývojové prostředí Eclipse na operačním systému Fedora 14, takže lze předpokládat, že překladač může obsahovat drobné nedostatky jako místy zapomenuté pouze jednoznakové odřádkování nepříjemně citelné ve Windows při zkoumání logovacího souboru. Kritické chyby jako neošetřená lomítka v cestách adresářů by ale TulToAsm51 snad obsahovat neměl.

2 PŘEKLADAČ OBECNĚ

V oblasti IT se překladačem rozumí program, který je schopen převést vstupní zdrojový text přímo do strojových instrukcí cílového procesoru. Vnitřní konstrukci snad blíže osvětlí ilustrace.



Ilustrace 1: Schéma moderního překladače

Z tohoto schématu je patrné, že jen samotný překladač sestává ze tří částí – dvou povinných a jedné volitelné. Front-end se stará o převod vstupního textu na mezikód, v případě TulToAsm51 derivační strom (existují i jiné varianty, např post-fixový tvar), musí postihnout lexikální i syntaktickou analýzu. Změnou této části lze přepsat překladač pro jiný vstupní jazyk. Část middle-end slouží k optimalizaci tohoto mezikódu, TulToAsm51 tuto část zatím neobsahuje. Překlad do cílového strojového zajistí back-end.

2.1 Druhy překladačů

Překladače lze rozdělit podle dvou kritérií. Jednak na rychlé jednoprůchodové bez sebemenší optimalizace a důsledné víceprůchodové, za druhé na typické překladače a inteprety. Interprety vstupní zdrojový kód nekompilují, ale rovnou spouští za podpory svého mateřského prostředí.

3 TUL KÓD

Jazyk TUL se snaží pomocí co nejjednodušších syntaxí zpřístupnit základní konstrukce. Aby byl čitelný i na monochromatických displejích, používá pro psaní klíčových slov výhradně velká písmena. Též je nutno dodržet pravidlo velkých písmen na začátku jmen funkcí a malých písmen na začátku jmen proměnných. Pokud název proměnné sestává z více než jednoho písmene, je vhodné vyhnout se číslu na druhém místě, zabrání se tak záměně s přímou hodnotou (viz Čísla).

3.1 Chomského hierarchie

Noam Chomsky rozděluje všechny gramatiky do čtyř vnořených tříd, tedy typ 3 znamená speciální případ typu 2, stejně tak typ 2 patří i do typu 1 a všechny do typu 0.

3.1.1 Gramatiky typu 0

Jedná se o gramatiky obecného typu. Pravidla takovéto gramatiky jsou neomezená, tak jak jsou popsána v definici gramatik. Tyto gramatiky generují jazyky L_0 typu 0.

3.1.2 Gramatiky typu 1 (kontextové gramatiky)

Tyto gramatiky nazýváme kontextové. Pravidla takovéto gramatiky mají tvar: $\lambda A\mu \rightarrow \lambda\nu\mu$, kde A je neterminál, který může být přepsán na řetězec ν pouze v kontextu levého řetězce λ a pravého řetězce μ.

V kontextových gramatikách není možné přepsat neterminál na prázdný řetězec. Jedinou výjimku tvoří pravidlo $S \rightarrow \varepsilon$, pak se ale S nesmí vyskytnout na žádné pravé straně pravidel.

Intuitivně je zřejmé, že nemůže dojít ke zkrácení derivovaných řetězců. Můžeme tedy říci, že pokud existuje $\rho \Rightarrow \sigma$ pak $|\rho| \leq |\sigma|$.

Jazyky generované takovouto gramatikou pak nazýváme kontextové jazyky a obecně je značíme L_1.

3.1.3 Gramatiky typu 2 (bezkontextové gramatiky)

Pravidla těchto gramatik mají tvar $A \rightarrow \lambda$, kde A je neterminál a λ řetězec z $(N \cup \Sigma)^*$. Přepisování je možné bez ohledu na kontext.

Pokud gramatika obsahuje pravidlo $S \rightarrow \varepsilon$, pak se též nesmí S objevit na žádné pravé straně pravidel.

Jazyky této třídy gramatik nazýváme bezkontextové jazyky a značíme je L_2.

3.1.4 Gramatiky typu 3 (regulární gramatiky)

Všechna pravidla gramatik tohoto typu jsou ve tvaru $A \rightarrow aB$ nebo A → a, kde $a \in \Sigma$ a $A, B \in N$.

Samozřejmě i tyto gramatiky mohou obsahovat pravidlo $S \rightarrow \varepsilon$, pokud se S neobjeví na žádné pravé straně pravidel.

Tyto gramatiky nazýváme regulární a generují regulární jazyky označované obecně L_3.

Kvůli zachování přesnosti citováno z [3]. Dle této hierarchie lze pravidla jazyka TUL zařadit mezi bezkontextové gramatiky, tedy typu 2, jak bude ukázáno v kapitole Parser.

3.2 Projekt a funkce

Projekt v jazyce TUL tvoří adresář se soubory s příponou .tul. Každý z nich může obsahovat jednu nebo více funkcí a v celém projektu překladač očekává právě jednu funkci Main. Ta je implicitně volána jako první a neměla by obsahovat vstupní ani výstupní proměnné, nicméně pokud by např. uživatel chtěl jedinou rekurzivní funkci (tedy s názvem Main), lze je nadefinovat a používat. Ve zdrojovém kódu se každá funkce uvozuje klíčovým slovem `NAME`, následně Jménem Funkce a středníkem.

Každá funkce se skládá z hlavičky a těla (bloku). Hlavička obsahuje deklarace všech vnitřních proměnných příslušné funkce. Kromě Main musí být každá fukce řízena jedním nebo více vstupy a poslat alespoň jeden výstup. Překladač pamatuje i na rekurzivní funkce, uživatel však musí vést v patrnosti, že operuje jen s jednou sadou proměnných. To platí i v případě rekurzivního volání více funkcí navzájem. Zabrání se tak nepředvídatelným chybám o nedostatku paměti.

3.3 Proměnné

Deklaraci vstupní proměnné uvozuje klíčové slovo `IN`, výstupní `OUT`, pomocné pak `HLP`, následuje jméno začínající malým písmenem a určení velikosti v bitech. Každá proměnná dostane svoje místo v paměti podle zadaného počtu bitů, přičemž se nabízí paleta velikostí od 1 do 32 bitů po mocninách dvou. Pokud se k číslu přidá ještě −, tak MSB této proměnné bude vyhrazen pro znaménko.

K portům a jejich pinům se přistupuje také přes hlavičku, a to označením `PORT` a `PIN`. Procesor 8051 disponuje standardně čtyřmi porty o osmi pinech, TulToAsm tudíž předpokládá 1-bajtovou velikost hodnot posílaných na port a 1-bitovou na pin. Uživatel však musí určit adresu portu, případně pinu (`PORT port P2; PIN pin P3.4;`). Lze tyto poté používat libovolně obousměrně na vlastní zodpovědnost, hodnota na portu se čte vždy přímo.

3.4 Příkazy a bloky

Blok znamená posloupnost příkazů ohraničenou klíčovým slovem "DO" a středníkem. Může se jednat o celé tělo funkce nebo jen podprogram větvícího či cyklotvorného příkazu.

TUL si vystačí se základními příkazy: přiřazení (`<proměnná> = <výraz>;`), větvení (`IF <podmínka> <blok> ELSE <blok>`) a cykly for (FOR <proměnná> = <výraz> : <výraz> <blok>) a while (`WHILE <podmínka> <blok>`).

3.5 Výrazy

Výraz v jazyce TUL může obsahovat kulaté závorky a znaménka `+`, `−`, `*`, `/` a `%` (značící zbytek po dělení). V jednom výrazu se nesmí provádět více než 9 operací `+` a `−` nebo `*`, `/` a `%`. Tento nedostatek odstraňuje vhodné použití závorek nebo přiřazení mezivýsledku do pomocné proměnné. Samozřejmě v TulToAsm funguje priorita početních výkonů.

3.6 Podmínky

Podmínky větvení `IF` nebo cyklu `WHILE` lze slučovat pomocí logických operátorů `AND`, `NAND`, `OR`, `NOR`, `XOR`, `NXOR`. Mezi operandy se vkládají znaménka `<`, `>`, `==`, `!=`, `<=`, `>=`. Takové porovnání může uživatel též nahradit 1-bitovou proměnnou (např `PIN`em), ne však přímo jednomístným binárním číslem.

3.7 Čísla

V TUL před každým absolutním číslem píše malé počáteční písmeno jedné ze čtyř dostupných soustav (b - binární, o - oktalová, d - decimální, h - hexadecimální), ve které jej uvažujeme. Například binární hodnotu true tedy vyjádříme jako `b1` (nelze ji však takto použít místo podmínky).

3.8 Příklad programu v TUL

Výše zmíněná kalkulačka se dvěma vstupními porty pro osmibitové operandy, dvěma piny pro zadání znaménka a jedním výstupním portem by se řešila následujícím způsobem. Do procesoru 8051 však v základní verzi vede pouze čtyřbajtová brána, tudíž výsledek násobení bude tvořit pouze jeho nižších 8 bitů.

Znaménko se bude zadávat takto:
00 - sčítání
01 - odčítání
10 - násobení
11 – dělení

Uvedený kód se snaží ukázat co nejvíce vlastností jazyka TUL najednou, v běžné praxi se takto jednoduchá aplikace snadno vtěsná do jediné funkce.

3.8.1 Kalkulačka v TUL

```
# Kalkulacka

NAME Main;

# deklarace operandu
   PORT a p1;
   PORT b p2;
# deklarace dvou pinu pro znamenko
   PIN zn1 p3.1;
   PIN zn2 p3.2;
# deklarace vystupniho portu
   PORT res p4;

DO
   IF (zn1) DO
      res = Nasdel(a, zn2 ,b).res;
   ;
   ELSEDO
      res = Sciode(a, zn2, b).res;
   ;
;

# Scitacka/Odecitacka

NAME Sciode;

# deklarace vstupu
   IN op1 8-;
   IN zn 1;
   IN op2 8-;
# deklarace vystupu
   OUT res 8-;

DO
   IF (zn) DO
      res = op1 - op2;
   ;
   ELSEDO
      res = op1 + op2;
   ;
;

# Nasobicka/Delicka

NAME Nasdel;

# deklarace vstupu
   IN op1 8-;
   IN zn 1;
   IN op2 8-;
# deklarace vystupu
   OUT res 8-;

DO
   IF (zn) DO
      res = op1 / op2;
   ;
   ELSEDO
      res = op1 * op2;
   ;
;
```

4 CHYBOVÁ HLÁŠENÍ

Ještě dříve než kýžený seznam assembleřích instrukcí se od překladače očekává upozornění na každou jednotlivou chybu, které se nepozorný programátor při bezhlavém bastlení zdrojového textu dopustil. Všechny potřebné nástroje pro nadávání uživateli obsahuje balíček `errors`. Každá instance třídy `Error` obsahuje číslo pořadí a hlášku, jenž bude vypsána přímo při vytvoření a při volání metody report statické třídy `Errors`. Bližší informace o chybě se vkládají jako parametr chybových pseudo-konstruktorů (viz přílohu - balíček `errors`, třída `Errors`, metody `quit`, `msg`, `expect`, ...). Tento způsob ani v nejmenším neomezuje svobodu projevu při popisu vzniklé události a zároveň šetří komentáře v java-kódu. Vzhledem k těmto dvěma výhodám bylo vhodné zařadit do balíčku i třídu `Messages` (viz přílohu - balíček `errors`, třída `Messages`), kterou snad netřeba více komentovat.

Některé chyby se podobají a lze je tedy určitým způsobem rozdělit na různé typy. V případě TulToAsm se jedná o šest kategorií podle smyslu a umístění daného nedopatření. Třída `Errors` tedy umožňuje volání ukončovací, nebo též fatální, chyby metodou `quit`, tu volá scanner, takže počítá s umístěním právě na jeho aktuální pozici. Neurčitou hlášku obstarává metoda `msg`, pokud se jedná pouze o varování, použije se `warn`. Pokud překladač očekává něco, co nenachází, zavolá metodu `expect`. Všechny tři fungují pro scanner i parser. Před samotným překladem derivačního stromu je potřeba zkontrolovat vzájemné volání funkcí kvůli parametrům - jestli souhlasí počty vstupů a zda existuje požadovaná výstupní proměnná. Když se cokoliv zde (viz přílohu - balíček `callTable`, třída `CallTab`, metoda `report`) pokazí, zavolá se metoda `Errors.postCtrl`. Zbývá poslední typ chyby, a tím je jakýkoliv nedostatek ve výstupu, například předvídatelné podtečení zásobníku nebo nedostatek vnitřní paměti cílového procesoru. Řeší je metoda `asm`, odkazující se na konkrétní řádek výstupního assembleřího kódu.

Metoda report vypíše všechny dosud uložené chybové hlášky. Volá se po ukončení parsování a po ukončení nebo selhání překladu. Vše neošetřené ošetřuje

výpis stromu odkazů volání funkcí v java kódu a milé pobídnutí uživatele k opravení překladače.

5 SCANNER

Jako asi každý překladač textových kódů i TulToAsm51 zpracovává zprvu nesmyslnou změť ASCII znaků. První pokusy vedly cestou rozdělování zdrojového kódu na symboly po mezerách. Není to dobrý nápad. Přinejmenším na mezery před středníky na koncích příkazů by si asi většina programátorů nerada zvykala. Proto se musí vstupní text nejprve znak po znaku rozdělit na symboly.

V TulToAsm51 vypadá například pro vstupní řádek `prom = 1+pom°;` výstup ze scanneru takto: `s_Var prom; s_Eq; s_Size 1; s_Plus; s_Var pom;` (Chyba) `s_Sem;`. Stupeň ° v TUL nic neznamená, tudíž bude vynechán a uživatel bude obdařen chybovou hláškou o neočekávaném znaku. Nechť si čtenář ráčí povšimnout symbolu `s_Size 1`. Vzhledem k tomu, že jazyk TUL zvládá výpočty s operandy v různých číselných soustavách potřebuje absolutně zadaná čísla nějak rozpoznat. Jak již bylo řečeno, hodnoty lze zadávat v binární (`b0`), oktalové (`o15`), decimální (`d29`) nebo hexadecimální (`hAF`) soustavě. Jen tak napsaná jednička tudíž zbyla na určení velikosti proměnné (1 bit). Tato okolnost ukazuje, že scanner rozhodně neřeší správnou posloupnost symbolů, pouze rozdělí text na slova, která zná.

5.1 Jak to funguje?

Vzhledem k relativní jednoduchosti syntaktické analýzy oproti dalším částem si překladač vystačil s jednou nosnou metodou `Source.scan` - obsahuje ji celkem pochopitelně balíček `scanner` - která vytvoří seznam všech symbolů (třídy `Symbol`) z jednoho souboru. Nepatrná výhoda takového postupu tkví ve vyšší rychlosti algoritmu, pokud se například ve vstupním souboru objeví znaky `NA`, může se jednat o `NAME` nebo `NAND`. Kdyby se volaly nějaké funkce `readName` a potom `readNAND`, muselo by se scanner v případě `NAND` vracet o dvě písmena a číst je znovu v metodě `readNAND`. To znamená, že pokud se přidává nové klíčové slovo začínající stejně

jako nějaké jiné, mělo by se jeho rozpoznání zařadit k jemu podobným. Každé přečtené slovo také implikuje hlášku o jeho nalezení, v některých případech je scanner schopen opravit i překlepy (např. `s_Size` začínající trojkou může nabývat hodnoty pouze 32), o čemž uživatele samozřejmě také informuje. I přes ukrutnou délku metody `scan` však lze jejímu algoritmu snadno porozumět pouhým letmým zhlédnutím java kódu.

Instance třídy `Symbol` obsahuje řetězec označující daný symbol (např. `s_Var`), hodnotu (`prom`) - ta může být i prázdná - a boolean funkce ke zjištění typu symbolu (`isName`, `isSize`, `isVar`...).

Ve stejném balíčku se nachází též statická knihovna `Info` obsahující informace o aktuálně čteném souboru a řádce - tenkrát ji ještě nebylo kam inteligentně umístit.

5.2 Klíčová slova

Jak již bylo řečeno, všechna klíčová slova se píší velkými písmeny, jinak nebudou rozpoznána (resp. budou považována za názvy proměnných).

Protože právě čtete kapitolu Scanner - Jak to funguje? uvádím jejich seznam v pořadí, v jakém se testují a doporučuji zároveň nahlížet do java-kódu (příloha - balíček `scanner`, třída `Source`, metoda `scan`). Zde by se překladač nechal ještě optimalizovat na vyšší rychlost seřazením podmínek podle odhadovaného počtu výskytů jednotlivých slov ve vstupním souboru. Ale na výstup v assembleru si může programátor bez větší újmy počkat, spíše ocení, když nebude překladačem vyplivnutý kód příliš týrat jeho oblíbenou hračku astronomickými nároky na programovou a operační paměť.

Pokud scanner najde klíčové slovo `NAME`, zvýší nadřazený `parser.Reader` počet funkcí o jednu. Důvod snad osvětlí kapitola Parser. Soubor se čte po řádcích, což umožňuje hbitý přeskok jednořádkových komentářů.

`#`	- jednořádkový komentář, co je za křížkem se nečte
`AND`	- logický součin používaný v podmínkách
`DO`	- začátek programového bloku (tělo funkce, podprogram cyklu nebo podmínky)

`ELSE` - else větev podmiňovacího příkazu `IF`

`FOR` - označení cyklu s inkrementovanou proměnnou

`HLP` - označení pomocné proměnné

`IN` - označení vstupní proměnné

`IF` - logická podmínka if

`NAME` - přiřazení jména funkce

`NAND` - negovaný logický součin

`NOR` - negovaný logický součet

`NXOR` - negovaný exkluzivní logický součet

`OR` - logický součet

`OUT` - označení výstupní proměnné

`PIN` - označení 1-bitové proměnné připojené na pin (fyzicky vstupně - výstupní, v programu jako pomocná)

`PORT` - označení 1-bajtové proměnné připojené na port (fyzicky vstupně - výstupní, v programu jako pomocná)

`XOR` - exkluzivní logický součet

`WHILE` - cyklus s podmínkou

5.3 Další čitelná slova

Už tady si člověk uvědomí, že nelze rozdělit vstupní text podle mezer. Díky striktním pravidlům pro psaní velkých a malých písmen by se dalo přidat automatické doplňování klíčových slov při překlepech, ale jak každý softwareista dobře ví, program se dá ladit donekonečna.

`JmenoFunkce` - název metody začínající velkým písmenem

`b<číslo>` - binární hodnota (0 nebo 1)

`o<číslo>` - oktalová hodnota (číslice 0..7)

`d<číslo>` - decimální hodnota (číslice 0..9)

`h<číslo>` - hexadecimální hodnota (číslice 0..F)

`p<číslo>` - adresa portu (0..3)

`p<číslo>.<číslo>` - adressa pinu (0..3, 0..7)

`jmenopromenne1` - jen z malých písmen následovaných čísly

`<číslo>` - určení velikosti unsigned proměnné v bitech (1..32)

`<číslo>-` - určení velikosti signed proměnné v bitech (1..32)

`+` - aritmetický součet

`-` - aritmetický rozdíl

`*` - aritmetický součin

`/` - celočíselné dělení

`%` - zbytek po celočíselném dělení

`;` - univerzální ukočovač

`.` - tečka k odkázání na výstup volané funkce

`,` - rozdělovač parametrů volané funkce

`:` - označení rozsahu inkrementace proměnné pro cyklus for

`(` - levá závorka (víceúčelová)

`)` - pravá závorka (víceúčelová)

`!=` - nerovnítko pro podmínky

`==` - rovnítko pro podmínky

`=` - přiřazovací příkaz

`<` - menší než (pro podmínky)

`<=` - menší nebo rovno (pro podmínky)

`>` - větší než (pro podmínky)

`>=` - větší nebo rovno (pro podmínky)

Vše ostatní scanner zahodí.

Metoda report vytvoří soubor s příponou .tulsym obsahující posloupnost všech nalezených symbolů včetně jejich hodnot. Stejně jako v případě výstupu do assembleru, i zde se pamatuje na Windows a jejich nesmyslné dva znaky pro odřádkování (`<cr><lf>`).

5.4 Posloupnost symbolů

Například funkce Main výše popsané kalkulačky tedy bude vyjádřená v terminálních symbolech vypadat následujícím způsobem.

```
s_Name, s_Id Main, s_Sem
   s_Port, s_Var a, s_Addr p1, s_Sem
   s_Port, s_Var b, s_Addr p2, s_Sem
   s_Pin, s_Var zn1, s_Addr p3.1, s_Sem
   s_Pin, s_Var zn2, s_Addr p3.2, s_Sem
   s_Port, s_Var res, s_Addr p4, s_Sem
s_Do
   s_If, s_LBr, s_Var zn1, s_RBr, s_Do
      s_Var res, s_Eq, s_Id Nasdel, s_LBr, s_Var a, s_Comma, s_Var zn2,
         s_Comma, s_Var b, s_RBr, s_Dot, s_Var res, s_Sem
   s_Sem
   s_Else, s_Do
      s_Var res, s_Eq, s_Id Sciode, s_LBr, s_Var a, s_Comma, s_Var zn2,
         s_Comma, s_Var b, s_RBr, s_Dot, s_Var res, s_Sem
   s_Sem
s_Sem

s_Name, s_Id Sciode, s_Sem
   s_In, s_Var op1, s_Size 8-, s_Sem
   s_In, s_Var zn, s_Size 1, s_Sem
   s_In, s_Var op2, s_Size 8-, s_Sem
   s_Out, s_Var res, s_Size 8-, s_Sem
s_Do
   s_If, s_LBr, s_Var zn, s_RBr, s_Do
      s_Var res, s_Eq, s_Var op1, s_Minus, s_Var op2, s_Sem
   s_Sem
   s_Else, s_Do
      s_Var res, s_Eq, s_Var op1, s_Plus, s_Var op2, s_Sem
   s_Sem
s_Sem
```

```
s_Name, s_Id Nasdel, s_Sem
   s_In, s_Var op1, s_Size 8-, s_Sem
   s_In, s_Var zn, s_Size 1, s_Sem
   s_In, s_Var op2, s_Size 8-, s_Sem
   s_Out, s_Var res, s_Size 8-, s_Sem
s_Do
   s_If, s_LBr, s_Var zn, s_RBr, s_Do
      s_Var res, s_Eq, s_Var op1, s_Div, s_Var op2, s_Sem
   s_Sem
   s_Else, s_Do
      s_Var res, s_Eq, s_Var op1, s_Mul, s_Var op2, s_Sem
   s_Sem
s_Sem
```

6 PARSER

Když překladač rozezná ve vstupním textu všechna slova, může se začít zabývat významem jejich posloupnosti - syntaktickou analýzou. Zatímco pro scanner (lexikální analýzu) se Java může zdát jako nejnepřátelštější platforma u parseru je tomu naopak. Možnost vzájemné vícevrstvě rekurzivní provázanosti tříd vytváří ideální podmínky pro implementaci mechanismu skládání derivačního stromu (viz Ročníkový projekt - Teorie tvorby překladačů vyšších programovacích jazyků - kap. 7.2.). Ten se řídí níže uvedenými pravidly. Pozorný čtenář si všimne velice jednoduchého převodu do jazyku Java pomocí volání konstruktorů podřízených členů v konstruktorech členů nadřízených (viz přílohu balíček `parser` - kromě statické metody `Reader`, ta bude popsána dále).

6.1 Pravidla jazyka TUL v Backus - Naurově formě

Jak vidno, celý <projekt> parser rozkládá na menší a menší členy (neterminály) až se dostane na terminální symboly (v BN formě se ohraničují uvozovkami). Zde se skýtá ještě jedna možnost, jak chápat a programovat mechanismus syntaktické analýzy. Lze též číst slovo po slovu od začátku a stavět uzly stromu vždy nad každou nalezenou smysluplnou posloupností uložených symbolů. Tento přístup se však stává

v javovském myšlení krajně nepřehledným, proto byl pro TulToAsm51 zvolen obrácený. Jinými slovy parser vychází z předpokladu, že uživatel chce přeložit v první řadě celý <projekt>, projekt bude patrně obsahovat nějaké funkce, každá funkce se skládá z hlavičky a těla atd., až se dostane přímo k symbolům, které pro něj scanner tak pěkně vyluštil ze zprvu nesmyslné změti ASCII znaků.

```
<project> ::= <function>*
<function> ::= <head> <block>
<head> ::= <title> [<input>*] [<output>*] [<helper>*] [<pin>*]
           [<port>*]
      // funkce bez vstupů nebo bez výstupů kromě Main však v TUL
nedávají smysl – nelze se na ně odkazovat
<block> ::= "s_Do" [<command>*] "s_Sem"
<command> ::= (<equation> "s_Sem") | <for> | <if> | <while>
<equation> ::= "s_Var" "s_Eq" <expression>
<for> ::= "s_For" <equation> "s_Col" <expression> <block>
<if> ::= "s_If" <condition> <block> ["s_Else" <block>]
<while> ::= "s_While" <condition> <block>
<title> ::= "s_Name" "s_Id" "s_Sem"
<input> ::= "s_In" "s_Var" "s_Size" "s_Sem"
<output> ::= "s_Out" "s_Var" "s_Size" "s_Sem"
<helper> ::= "s_Hlp" "s_Var" "s_Size" "s_Sem"
<pin> ::= "s_Pin" "s_Var" "s_Addr" "s_Sem"
<port> ::= "s_Port" "s_Var" "s_Addr" "s_Sem"
<expression> ::= <term> [<addOp> <term>]*
<term> ::= <number> | ("s_LBr" <expression> "s_RBr") [<mulOp> <number>
           | ("s_LBr" <expression> "s_RBr")]*)]
<number> ::= "s_Bnum" | "s_Onum" | "s_Dnum" | "s_Hnum" | "s_Var" |
             <call>
<condition> ::= <comparation> [<logOp> <comparation>]*
<comparation> ::= (<expression> <cmpOp> <expression>) | <pin> |
                  "s_Var"
      // pouze jednobitová proměnná může být použita místo porovnání
<cmpOp> ::= "s_IsEq" | "s_IsNEq" | "s_IsSm" | "s_IsGr" | "s_IsSmEq" |
            "s_IsGrEq"
<call> ::= "s_Id" "s_LBr" <expression> ["s_Comma" <expression>]*
           "s_RBr" "s_Dot" "s_Var"
```

```
// Legenda:
//    <cosi>      - neterminál
//    "cosi"      - terminál (symbol)
//    ::=         - přiřazení
//    ()          - prosté závorky
//    []          - může obsahovat
//    *           - jednou nebo vícekrát
//    |           - nebo (& se předpokládá)
```

6.2 Jak to funguje?

Statická třída `Reader` slouží jako rozhraní mezi scannerem a parserem. Důvod takových předávacích mechanismům mezi jednotlivými částmi překladače spočívá v možnosti jejich výměny za jiný modul. Pokud se někomu nelíbí navržená syntaxe jazyka TUL, přepíše si jen metodu `scanner.Source.scan` (viz kapitolu Scanner), pokud si zrovna nehraje s 8051, vytvoří si překladač třeba pro Siemens Logo "pouhou" rekonstrukcí balíčku `compiler` (bude zmíněn později).

Nyní je doporučeno zároveň sledovat přílohu: balíček `parser`, třída `Reader`, všechny její části budou blíže osvětleny. Scanner zpracovával přímo vstupní soubor, což znamená, že jeho výstupem bude sada symbolů ukrývající (v ideálním případě) jednu nebo více funkcí. A vzhledem k tomu, že projekt v jazyce TUL určuje složka obsahující, pokud má přeložený program nějak fungovat, jeden nebo více souborů *.tul, z nichž právě jeden ukrývá funkci nazvanou Main, musí se všechny nalezené sady symbolů dále popsaným způsobem sloučit. To má na starosti metoda `scanProject`. Celkem pochopitelně naplní pole `symbols`, které metoda `makeTree` převede dle výše uvedených pravidel do formy derivačního stromu (instance třídy `Project`). K tomu stačí zavolat konstruktor nejvyššího objektu `project` (viz přílohu - balíček `parser`, třída `Project`). Odtud se pak postupně sestaví všechny funkce, každá si zavolá konstruktor hlavičky a těla, hlavičky si vytvářejí nadpis a proměnné atd. dle pravidel gramatiky TUL, snad netřeba popisovat zvlášť jednotlivé třídy.

V některých konstruktorech se však nesmí zapomenout na plnění příslušných tabulek. Při sestavování volání `Call` je nutné uvést záznam do tabulky volání, stejně tak při deklaraci proměnných (třídy `Input`, `Output`, `Helper`, `Pin` a `Port`) se musí tyto uložit do tabulky. Třída `Number` ještě kromě předpokládaného uvedení záznamu o použití proměnné do tabulky obstarává přepočet velikosti absolutně zadaných čísel do desítkové soustavy pro snazší vyšetření paměťového objemu výsledku výrazu.

Protože parser provádí syntaktickou analýzu, podává také chybová hlášení o špatné posloupnosti symbolů. Ideálně zde slouží typické větvení `if`/`else if`/.../`else`, kdy `else` blok pošle zprávu s výběrem očekávaných symbolů (zavolá metodu `Errors.expect`). Pokud uživatel zapomene na jeden formální symbol, příslušné nadávky se samozřejmě dočká, ale úspěšnému překladu tato chyba nebrání. Například příkaz `a = Funkce(par1, par2).vystup;` lze celkem beztrestně zprznit na `a = Funkce(par1,par2vystup` (viz přílohu - balíček `parser`, třída `Call`). Takový případ by někdo mohl považovat za tzv. varování, ale v tom případě by se měl programování radši obloukem vyhnout. TulToAsm51 „varuje“ uživatele (zatím) pouze před typovou nesrovnalostí. Z ladicích důvodů zůstává však cílem překladače vygenerovat za každou cenu třeba i nefunkční assembleří kód (v parseru tedy derivační strom - plnohodnotnou instanci třídy `Project`).

Po dokončení parsování se uživateli zobrazí všechny proměnné, volání funkcí a chyby - tento výpis se provede také po dokončení kompilace, je to z toho důvodu, že pokud se nedostane na samotný překlad compilerem, uživatel alespoň nemusí složitě hledat v "logu", co udělal špatně (ale samozřejmě může).

7 TABULKA PROMĚNNÝCH

Pokud má jazyk, potažmo překladač, nabídnout uživateli pestrou paletu různých formátů jeho záludných proměnných, musí nezbytně nějaký orgán vést záznamy o všech deklaracích, použitích, v pokročilejší fázi i přímo fyzických adresách v RAM mikrokontroleru. TulToAsm51 zatím nepodporuje alokaci externí paměti, nicméně lze tento mechanismus v případě potřeby doplnit do java-kódu (Musel by se zavést nový

typ proměnné, kupříkladu `EXT`, s tím, že by se používala stejně jako `HLP`. Nejsložitější bude využití ukazatele DPTR - do externí paměti - ve výstupním assembleřím kódu).

V příloze - balíček `varTable`, jistě si na první pohled jistě čtenář všimne, že zde panuje přísná hierarchie mezi třemi obsaženými třídami. Přeložitelný projekt jazyka TUL, jak již bylo předesláno v kapitole Jazyk TUL, sestává z jedné nebo více funkcí, tudíž je nutno rozlišovat, ke které funkci patří která proměnná. Tuto informaci uchovávají řádky (třída `VarTabLine`) zapouzdřené v tabulce (třída `VarTab`) příslušné funkce projektu (v tomto případě reprezentuje statická třída `VarTabs`, přes kterou se k TUL proměnným v java-kódu přistupuje).

7.1 VarTabLine

Každý řádek tabulky obsahuje krátké a dlouhé jméno proměnné (dlouhé bude použito v assembleřím kódu, skládá se z názvu funkce, podtržítka a jména krátkého), pole adres - pole protože 8051 umí jen bity a bajty, word se už musí uložit do dvou bytů - dálé se zde nalézá velikost, minimum, maximum, typ (`IN`, `OUT`, `HLP`, `PIN` nebo `PORT`) a bitové informace o čtení, deklaraci a vlivu na I/O.

8 TABULKA VOLÁNÍ

Tabulka volání funguje podobně jako tabulka proměnných, obsahuje však pouze dvě třídy. `CallTabLine` (viz přílohu - balíček `callTable`) znázorňuje jeden řádek tabulky volání a obsahuje určení místa nalezení, parametry ve formě výrazů včetně jejich počtu, název odkazované funkce a požadovaného výstupu, konstruktor, který všechny tyto hodnoty naplní, a nutné gettery. Řádek se vytváří metodou `call` třídy `CallTab`, ta ještě obsahuje proceduru výpisu všech volání včetně chybových hlášení.

9 OPTIMALIZACE

Mezi parser a compiler patří obvykle ještě, povětšinou poměrně složitý, nástroj pro optimalizaci derivačního stromu. Ten ovšem zatím chybí, lze jej však celkem bez obtíží implementovat do dalšího balíčku a zavolat jeho spouštěcí příkaz v metodě `Compiler.run` (viz přílohu). Jednalo by se jen o slučování stejných podvýrazů, rozkládání cyklů s pevným počtem opakování, zařazení chudších zřídka volaných funkcí přímo do míst volání. Problémy s nevyužitou alokovanou pamětí jsou snad v rámci kompilátoru vyřešeny obsazováním pouze nejnutnějších registrů a adres vnitřní RAM. Optimalizátor by měl vygenerovat opět derivační strom, jen přeuspořádaný, aby ho kompilátor dokázal zpracovat.

10 ROZPOZNÁNÍ OPERAČNÍHO SYSTÉMU A GLOBÁLNÍ NASTAVENÍ

Multiplatformnost překladače TulToAsm51 zajišťuje jediná statická třída `Settings` (viz přílohu - balíček `globalCtrl`, třída `Settings`). Linux od Windows rozezná úplně poslední metoda `chooseLinWin` velmi jednoduše testem existence root adresáře `"/"`, při kladném výsledku nastaví metodou `setLin` dopředné lomítko (slash) pro zadávání cest souborů a jednoduché odřádkování (lineFeed) `"\n"`. V opačném případě se metodou `setWin` uloží lomítko zpětné a dvouznakové odřádkování `"\r\n"`. Dále tato třída obsahuje název a cestu projektu, potřebné gettery a metodu `setProject`, které se předá cesta uvedená v parametru programu.

11 COMPILER

Nyní musí ještě takový krásný derivační strom pochopit také PLC. Compiler mu jej tedy rozloží na kusy, které lze vcelku nahradit assembleřími instrukcemi. Jedenapadesátkový dialekt umožňuje krátké a dlouhé nepodmíněné skoky s návratem, což skýtá jedinečnou příležitost v podstatě opsat TUL kód do Assembleru 51. Funkce

se definují každá zvlášť, ohraničí návěstím shora a klíčovým slovem `RET` zespoda a volají se pomocí `LCALL` (viz přílohu - Assembler 8051: Instrukční sada).

Záměrem bylo programovat kompilátor podobným způsobem jako parser přes konstruktory celků, které se snadno interpretují v assembleru, potažmo ve strojovém kódu. 8051 však nenabízí vymoženost moderních silných procesorů - standardní vícebajtové operace jako například `ADD32`. Sčítání, odečítání, násobení i dělení se tedy musí vyřešit pomocnými „kalkulačkami“. Ty se tvoří pro každou operaci zvlášť, což při větším množství „double-wordových“ proměnných ve vstupním kódu způsobuje značné kynutí kódu výstupního, na druhou stranu se šetří daleko cennější operační pamětí. Procesor též ve výsledku odbaví méně instrukcí, než kdyby musel dlouze skákat na předdefinované kalkulačky, nebo plnit zásobník nulami do počtu bajtů nejvyššího z operandů. Díky těmto mechanismům se může zdát statická třída `compiler.Compiler` zprvu trochu nepřehledná, ale v podstatě se nejedná o nic složitého, jak ukáží kapitoly Sčítačka, Odčítačka, Násobička a Dělička.

11.1 Jak to funguje?

Jádro celého procesu překládání tedy sídlí ve třídě `Compiler`, v ideálním případě obsahuje všechny jazykem TUL zadané aritmetické operace. Stal se zázrak, jestli příloha ukrývá i odčítačku, násobičku a děličku. Pokud ano, každou představuje jedna třída v balíčku `compiler`, všechny dostanou svojí vlastní kapitolu. Dále vzhledem k vícebajtovým mezivýsledkům zde se nalézají i metody podobné instrukcím assembleru jako `inc`, `dec`, `and`, ..., `pop`, `push`, ... Ke vkládání do zásobníku slouží ještě další dvě funkce: `pushA` a `pushSA`. Ty obstarávají vložení obsahu střadače (`A`) nebo mezivýsledků včetně `A` do zásobníku. Všem výše zmíněným metodám se předávají parametry v podobě polí řetězců, což zároveň řeší rozdílnost velikostí operandů. Vytažení hodnoty ze zásobníku samozřejmě počítá s prostornější cílovou pamětí a doplňuje tedy bajty navíc nulami.

Dále třída `Compiler` obstarává vložení jedné assembleří instrukce do pole `lines` a výpis tohoto pole na konzoli i do souboru. Slouží též jako startér celého překladače (metoda `run`). Vybere hostitelský operační systém, zavolá syntaktickou

analýzu, která si sama zařídí pomocí scanneru rozdělení vstupního textu na symboly. Z ladicích důvodů se uvolňují všechny registry a alokuje pomocný double-wordový akumulátor. Pod obklopením `try-catch` na jakoukoli neošetřenou výjimku se spustí pokus o vytvoření hlavičky a těla *.asm souboru. V případě neúspěchu se zobrazí výpis všech v průběhu překladu zjištěných chyb a rodokmen volání kritické metody v java - kódu doprovázený milým popudem k odbornému zásahu. Stále však platí zásada vytvořit alespoň nějaký výstupní kód - další varování před jeho použitím, nedej Bože se zapojením robota.

Statická třída `Loops` se stará jak o jedinečnost jmen návěstí, tak o obsazování a uvolňování registrů. To z toho důvodu, že právě registry se zatím, využívají jako pomocné proměnné v podmínkách cyklů a přeskoků (`IF`, `FOR`, `WHILE`). V budoucnosti by se nechal přidat ještě mechanismus na přepínání mezi bankami registrů pomocí nastavování bitů `RS0` a `RS1` stavového slova `PSW`. Aktuální obsazenost registrů se uchovává v osmibitovém poli `regUsage` a použitá jména návěstí v poli řetězců `loopNames` na jehož další prázdnou pozici ukazuje `loopPtr`. Zatím zde tedy bydlí metody s výmluvnými názvy `getLoopName`, `getFreeReg`, `freeReg` a `freeAllRegs`.

11.2 Sčítačka

Jak již bylo řečeno, vzhledem k tomu, že jazyk TUL samozřejmě počítá i se „signed“ proměnnými, musí se jim přizpůsobit i všechny operace. Konkrétně sčítání lze rozdělit na čtyři specifické případy dle `signed` vlastnosti operandů.

11.2.1 Případ unsigned + unsigned

Nejjednodušší situaci, tedy sčítání dvou zcela jistě nezáporných čísel řeší klasické sčítání – prvních bajtů bez přenosu (`ADD`) a vyšších s přenosem (`ADDC`) – samozřejmě s využitím střadače `A`. Ať už došlo k přetečení při sčítání nejvyšších bajtů nebo ne, uloží se do zásobníku i bit přetečení `C`, protože nadřazený výraz může pojmout i prostornější výsledek. Plnění zásobníku je obstaráno mimo

metody `push`, `pushA` či `pushSA`, neboť zrovna zde vychází daleko lépe ukládat jednotlivé bajty výsledku ihned po sečtení a ne až po celé operaci najednou.

11.2.2 Případ signed + unsigned

Pokud jeden z operandů může nabývat záporné hodnoty, musí se do sčítačky implementovat rozpoznání záporného čísla – v Assembleru se nabízí odečtení nejvyššího bitu (tedy hexa-hodnoty `#80h`) s kontrolou přetečení. Ve výstupním kódu se pak objeví `JNC` přeskok sčítání na odečítání. Při odečítání je pak nutno získat absolutní hodnotu prvního operandu, na to stačí jej znegovat (`CPL A`). V tomto případě se používá pomocný akumulátor, který se při záporném výsledku zneguje a do MSB nejvyššího bajtu (střadače `A`) se vloží znaménko.

11.2.3 Případ unsigned + signed

Případ obrácený, tedy první operand určitě nezáporný, druhý `signed` se řeší obdobně, jen se operandy prohodí.

11.2.4 Případ signed + signed

Nejobecnější situace vyžaduje ve výstupním kódu ošetření všech předchozích možností a navíc případu zápornosti obou operandů. Ten je ošetřen klasickým součtem absolutních hodnot a následným převrácením výsledku.

11.3 Odčítačka

Odečítání lze provést též za pomoci sčítačky, jen s poupraveným pořadím operandů v jednotlivých případech. TulToAsm51 v0.84 – build 12.05.17 tuto funkci z důvodu nedostatečně promyšlené struktury aritmetických operací ještě neposkytuje. Sčítačku bude nutno rozdělit na jednotlivé úkony dle znamének operandů, aby nevznikal zbytečně duplicitní java-kód.

11.4 Násobička

Násobení se liší od všech ostatních aritmetických operací v poměru velikostí operandů a výsledku. Ten totiž může nabývat rozměrů součinu | `op1` | * | `op2` |,

tedy například výsledek po násobení 32-bitového čísla 16-bitovým obsadí 6 bajtů (48 bitů), tudíž se mimo jiné nevejde do žádného z předdefinovaných datových typů. Přesně z těchto důvodů (přetečení mezivýsledků přes podporovanou hranici 4 bajtů) nepovažuje TulToAsm51 tzv. Type-mismatch za fatální chybu.

11.5 Dělička

Dělička zachovává pro výsledek velikost prvního operandu, v případě celočíselného dělení, nebo druhého, pokud uživatele zajímá zbytek po dělení. TulToAsm51 bude v plné verzi podporovat obě tyto dvě operace – tedy bez plovoucí řádové čárky (desetiny však nebývají standardem ani u jazyků pro větší PAC jako např. Simatic S7-300, či GE-Fanuc RX3i).

Klasické celočíselné dělení se musí provádět od nejvyšších bajtů, v podstatě stejně jako v desítkové soustavě na papíře. V zásobníku však lze očekávat jednotlivé bajty operandů v opačném pořadí, tedy od nejméně významného. Celé operandy je tudíž nutno uložit do pomocné paměti, aby se dalo nejdříve přistupovat k bajtům nejvýznamnějším.

12 SEZNAM LITERATURY

[1] RNDr. VAVREČKOVÁ, Šárka, Ph.D.: *Tvorba překladačů*. Slezská univerzita v Opavě – Filozoficko-přírodovědecká fakulta – Ústav informatiky. Opava 2008. 218 stran. ISBN 978-80-7248-493-5

[2] Ing. MARTINEC, Tomáš, Ph.D.: *Napište si překladač 1. - 4. díl.* K7. Letní číslo 2005 – jarní číslo 2006. ISSN 1214-7370

[3] *Http://automata.howto.cz/hierarchie.html* [online]. 2012 [cit. 2012-05-17]. Dostupné z: http://automata.howto.cz/hierarchie.html

13 ZÁVĚR

Cílem této bakalářské práce bylo navrhnout vlastní programovací jazyk pro ovládání PLC s procesorem Intel MCS-51 a pro tento jazyk vytvořit překladač do příslušného assembleru.

Jazyk TUL se zakládá na principech známých z dlouhými léty prověřeného výukového jazyka Pascal. Tedy potřebná paměť pro proměnné se alokuje staticky ještě před začátkem samotného programu. Vzhledem k mizivému využití jiných datových typů než bitů a čísel však na rozdíl od Pascalu nepodporuje znakové, výčtové ani jiné nestandardní typy. Zatím také chybí možnost definice polí, která by uplatnění sice jistě našla, nicméně TulToAsm51 používá pouze vnitřní paměť procesoru, tudíž na rozmáchlé operace s ohromnými počty proměnných najednou stejně nezbývá prostor.

Překladač sám dokáže převést vstupní soubor v TUL kódování nejprve do posloupnosti symbolů a v ní objevit všechny lexikální chyby pomocí balíčku `scanner`. Poté balíček `parser` zařídí nad těmito symboly sestavení derivačního stromu a samozřejmě zjištění všech chyb syntaktické analýzy (tyto chyby se stejně jako lexikální odkazují přímo na řádek vstupního textu). Posledním nástrojem nutným k úspěšnému překladu zůstává kompilátor (balíček `compiler`). Ten se stará o rozložení vzniklého derivačního stromu přímo do instrukcí Assembleru 8051 a v ideálním případě by neměl produkovat žádná chybová hlášení. Nicméně z důvodů ladění celého překladu kontroluje kompatibilitu dat vkládaných do a vyjímaných ze zásobníku.

Vývoj moderního plně funkčního překladače byť jednoduchého jazyka však obyčejně trvá nesrovnatelně déle, než aby se vešel do časového rozpočtu jedné bakalářské práce. Proto přiložený program TulToAsm51 v0.84 – build 12.05.17 stále obsahuje velké množství nedostatků, jako chybějící převod aritmetických operací (kromě sčítání) do Assembleru 8051, nepřítomnost optimalizátoru a linkeru a další neduhy.

PŘÍLOHA

Instrukční sada 8051

Speciální registry

Zkratka	Adresa	Význam
ACC*	0E0h	akumulátor
B*	0F0h	B registr
PSW*	0D0h	stavový registr
SP	81h	ukazatel na zásobník
DPTR	82h,83h	ukazatel na data (DPL, DPH)
P0*	80h	port 0
P1*	90h	port 1
P2*	0A0h	port 2
P3*	0B0h	port 3
IP*	0B8h	řízení priority přerušení
IE*	0A8h	povolení/zákaz přerušení
TMOD	89h	řízení režimu časovače/čítače
TCON*	88h	řízení časovače/čítače
TH0	8Ch	vyšší slabika časovače/čítače 0
TL0	8Ah	nižší slabika časovače/čítače 0
TH1	8Dh	vyšší slabika časovače/čítače 1
TL1	8Bh	nižší slabika časovače/čítače 1
SCON*	98h	řízení sériového rozhraní
SBUF	99h	data sériového rozhraní
PCON	87h	řízení spotřeby

Instrukční soubor

Aritmetické operace

ADD	A,Rn	Přičte obsah registru k akumulátoru
ADD	A,direct	Přičte obsah adresy k akumulátoru
ADD	A,@Ri	Přičte obsah nepřímé adresy k akumulátoru
ADD	A,#data	Přičte přímá data k akumulátoru
ADDC	A,Rn	Přičte CY a obsah registru k akumulátoru
ADDC	A,direct	Přičte CY a obsah adresy k akumulátoru
ADDC	A,@Ri	Přičte CY a obsah nepřímé adresy k akumulátoru
ADDC	A,#data	Přičte CY a přímá data k akumulátoru
SUBB	A,Rn	Odečte CY a obsah registru od akumulátoru

```
SUBB  A,direct       Odečte CY a obsah adresy od akumulátoru
SUBB  A,@Ri          Odečte CY a obsah nepřímé adresy od akumulátoru
SUBB  A,#data        Odečte CY a přímá data od akumulátoru
INC   A              Zvětší obsah akumulátoru o 1
INC   Rn             Zvětší obsah registru o 1
INC   direct         Zvětší obsah adresy o 1
INC   @Ri            Zvětší obsah nepřímé adresy o 1
DEC   A              Zmenší obsah akumulátoru o 1
DEC   Rn             Zmenší obsah registru o 1
DEC   direct         Zmenší obsah adresy o 1
DEC   @Ri            Zmenší obsah nepřímé adresy o 1
INC   DPTR           Zvětší obsah ukazatele dat o 1
MUL   AB             Vynásobí registry A a B
DIV   AB             Dělí registr A registrem B
DA    A              Desítková úprava obsahu akumulátoru
```

Logické operace

```
ANL   A,Rn           Logický součin akumulátoru s obsahem registru
ANL   A,direct       Logický součin akumulátoru s obsahem adresy
ANL   A,@Ri          Logický součin akumulátoru s obsahem nepřímé
adresy
ANL   A,#data        Logický součin akumulátoru s přímými daty
ANL   direct,A       Logický součin obsahu adresy s akumulátorem
ANL   direct,#data   Logický součin obsahu adresy s přímými daty
ORL   A,Rn           Logický součet akumulátoru s obsahem registru
ORL   A,direct       Logický součet akumulátoru s obsahem adresy
ORL   A,@Ri          Logický součet akumulátoru s obsahem nepřímé
adresy
ORL   A,#data        Logický součet akumulátoru s přímými daty
ORL   direct,A       Logický součet obsahu adresy s akumulátorem
ORL   direct,#data   Logický součet obsahu adresy s přímými daty
XRL   A,Rn           Exclusive-OR akumulátoru s obsahem registru
XRL   A,direct       Exclusive-OR akumulátoru s obsahem adresy
XRL   A,@Ri          Exclusive-OR akumulátoru s obsahem nepřímé adresy
XRL   A,#data        Exclusive-OR akumulátoru s přímými daty
XRL   direct,A       Exclusive-OR obsahu adresy s akumulátorem
XRL   direct,#data   Exclusive-OR obsahu adresy s přímými daty
CLR   A              Nuluje obsah akumulátoru
CPL   A              Neguje obsah akumulátoru
```

RL	A	Rotace obsahu akumulátoru vlevo
RLC	A	Rotace obsahu akumulátoru a CY vlevo
RR	A	Rotace obsahu akumulátoru vpravo
RRC	A	Rotace obsahu akumulátoru a CY vpravo
SWAP	A	Vymění horní a dolní nibble obsahu akumulátoru

Přesun dat

MOV	A,Rn	Přesun obsahu registru do akumulátoru
MOV	A,direct	Přesun obsahu adresy do akumulátoru
MOV	A,@Ri	Přesun obsahu nepřímé adresy do akumulátoru
MOV	A,#data	Přesun přímých dat do akumulátoru
MOV	Rn,A	Přesun obsahu akumulátoru do registru
MOV	Rn,direct	Přesun obsahu adresy do registru
MOV	Rn,#data	Přesun obsahu nepřímé adresy do registru
MOV	direct,A	Přesun obsahu akumulátoru na adresu
MOV	direct,Rn	Přesun obsahu registru na adresu
MOV	dirct1,dirct2	Přesun obsahu adresy na jinou adresu
MOV	direct,@Ri	Přesun obsahu nepřímé adresy na jinou adresu
MOV	direct,#data	Přesun přímých dat na adresu
MOV	@Ri,A	Přesun obsahu akumulátoru na nepřímou adresu
MOV	@Ri,direct	Přesun obsahu adresy na nepřímou adresu
MOV	@Ri,#data	Přesun přímých dat na nepřímou adresu
MOV	DPTR,#data16	Přesun 16-bitových dat do ukazatele dat
MOVC	A,@A+DPTR	Přesun kódového bajtu z adresy (A)+(DPTR) do akumulátoru
MOVC	A,@A+PC	Přesun kódového bajtu z adresy (A)+(PC) do akumulátoru
MOVX	A,@Ri	Přesun obsahu externí paměti do akumulátoru (8-bit addr)
MOVX	A,@DPTR	Přesun obsahu externí paměti do akumulátoru (16-bit addr)
MOVX	@Ri,A	Přesun obsahu akumulátoru do externí paměti (8-bit addr)
MOVX	@DPTR,A	Přesun obsahu akumulátoru do externí paměti (16-bit addr)
PUSH	direct	Uložení obsahu adresy na zásobník
POP	direct	Obnovení obsahu adresy ze zásobníku
XCH	A,Rn	Výměna obsahu akumulátoru s obsahem registru

XCH	A,direct	Výměna obsahu akumulátoru s obsahem adresy
XCH	A,@Ri	Výměna obsahu akumulátoru s obsahem nepřímé adresy
XCHD	A,@Ri	Výměna nižších nibblů mezi obsahem nepřímé adresy a akumulátorem

Booleovské instrukce

CLR	C	Vynuluje CY
CLR	bit	Vynuluje bit
SETB	C	Nastaví CY
SETB	bit	Nastaví bit
CPL	C	Neguje CY
CPL	bit	Neguje bit
ANL	C,bit	Logický součin CY s bitem
ANL	C,/bit	Logický součin CY s negací bitu
ORL	C,bit	Logický součet CY s bitem
ORL	C,/bit	Logický součin CY s negací bitu
MOV	C,bit	Přesun bitu do CY
MOV	bit,C	Přesun CY do bitu

Větvení programu

ACALL	addr11	Volání podprogramu uvnitř 2K stránky
LCALL	addr16	Dlouhé volání podprogramu
RET		Návrat z podprogramu
RETI		Návrat z přerušení
AJMP	addr11	Skok uvnitř 2K stránky
LJMP	addr16	Dlouhý skok
SJMP	rel	Krátký skok (relativní adresa)
JMP	@A+DPTR	Skok na adresu určenou (DPTR)+(A)

Podmíněné skoky

JC	rel	Skok při nastaveném CY
JNC	rel	Skok při nulovém CY
JB	bit,rel	Skok při nastaveném bitu
JNB	bit,rel	Skok při nulovém bitu
JBC	bit,rel	Skok a nulování při nastaveném bitu
JZ	rel	Skok při nulovém akumulátoru
JNZ	rel	Skok při nenulovém akumulátoru
CJNE	A,direct,rel	Skok při různém obsahu akumulátoru a adresy
CJNE	A,#data,rel	Skok při různém obsahu akumulátoru a přímých dat

```
CJNE  Rn,#data,rel   Skok při různém obsahu registru a přímých dat
CJNE  @Ri,#data,rel  Skok při různém obsahu nepřímé adresy a přímých
dat
DJNZ  Rn,rel         Zmenší obsah registru a skočí, je-li <> 0
DJNZ  direct,rel     Zmenší obsah registru adresy a skočí, je-li <> 0
NOP                  Žádná operace
```